AVF Control Number: AVF-IABG-069

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #900131I1.10270
TeleSoft
TeleGen2 Ada Development System
VAX/VMS host and MC68k target

Completion of On-Site Testing:
31 January 1990

Prepared By:
IABG mbH, Abt. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final 31 Jan 90 to 31 Jan 91

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Repo... TeleSoft
TeleGen2 Ada Development System, VAX VMS (Host) to MC68K (Target)
90013I1.10270

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, FEDERAL REPUBLIC OF GERMANY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-IABG-069

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

TeleSoft, TeleGen2 Ada Development System, Ottobrunn, West Germany, VAXserver 3800 under VAX/VMS Version 5.2 (Host) to Motorola MVME 133A-20 (MC68020)(Bare Machine), ACVC 1.10.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
299-01

Ada Compiler Validation Summary Report:

Compiler Name: TeleGen2 Ada Development System for
VAX to E68K, Version 4.0

Certificate Number: #900131I1.10270

Host: VAXserver 3800
under VAX/VMS Version 5.2

Target: Motorola MVME 133A-20 (MC68020)
(Bare machine)

Testing Completed 31 January 1990 Using ACVC 1.10

This report has been reviewed and is approved.

IABG mbH, Abt. SZT
Dr. S. Heilbrunner
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	X
DTIC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO).

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

IABG mbH, Abt. SZT
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler Name: TeleGen2 Ada Development System for
VAX to E68K, Version 4.0

ACVC Version: 1.10

Certificate Number: #900131I1.10270

Host Computer:

Machine:	VAXserer 3800
Operating System:	VAX/VMS Version 5.2
Memory Size:	32 MegaBytes

Target Computer:

Machine:	Motorola MVME 133A-20 (MC68020)
Operating System:	Bare machine
Memory Size:	1 MegaByte

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types SHORT_INTEGER, LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) Some of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses no extra bits for extra range. (See test C35903A.)

4) NUMERIC_ERROR is raised for integer comparison and membership tests except for smallest integer membership tests where no

exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR for a two dimensional array subtype where the large dimension is the second one. (See test C36003A)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that index subtype checks are made as choices are evaluated. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is not supported for library procedures or library functions. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This implementation creates a dependence between a generic body and those units which instatiate it. As allowed by AI-408/11, if the body is compiled after a unit that instantiates it, then that unit becomes obsolete.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO can be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

10) Temporary sequential files are given names and not deleted when closed. (See test CE2108A.)

11) Temporary direct files are given names and not deleted when closed. (See test CE2108C.)

12) Temporary text files are given names and not deleted when closed. (See test CE3112A.)

13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

15) More than one internal file can be associated with each external file for text files when reading only (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 296 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 10 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1132	2029	17	26	44	3377
Inapplicable	0	6	286	0	2	2	296
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	TEST CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	573	555	248	172	99	161	332	129	36	250	340	284	3377
N/A	14	76	125	0	0	0	5	0	8	0	2	29	37	296
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 296 tests were inapplicable for the reasons indicated:

> The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because they require a value of SYSTEM.MAX_MANTISSA greater than 32.

C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

CA2009C, CA2009F, BC3204C and BC3205D are not applicable because this implementation creates a dependence between a generic body and those units which instantiate it (See Section 2.2.i and Appendix F of the Ada Standard).

LA3004A, EA3004C, and CA3004E are not applicable because this implementation does not support pragma INLINE for library procedures.

LA3004B, EA3004D, and CA3004F are not applicable because this implementation does not support pragma INLINE for library functions.

CD1009C, CD2A41A..B (2 tests), CD2A41E and CD2A42A..J (10 tests) are not applicable because of restrictions on 'SIZE length clauses for floating point types.

CD1C04E is not applicable because this implementation does not support component clauses specifying more than 8 bits for boolean components of a record.

CD2A61I..J (2 tests) are not applicable because of restrictions on 'SIZE length clauses for array types.

CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because of restrictions on 'SIZE length clauses for access types.

CD4041A is not applicable because of restrictions on record representation clauses with 32 bit alignment.

CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

CE3102F is inapplicable because text file RESET is supported by this implementation.

CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

CE3102K is inapplicable because text file OPEN with OUT_FILE mode is supported by this implementation.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 10 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B71001E	B71001K	B71001Q	B71001W	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A (6 and 7)		

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the TeleGen2 Ada Development System for a computing system based on the same instruction set architecture was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the TeleGen2 System successfully passed all applicable tests, and it exhibited the expected behavior on all inapplicable tests. The applicant certified that testing results for the computing system of this validation would be identical to the ones submitted for review prior to validation.

3.7.2 Test Method

Testing of the TeleGen2 Ada Development System using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host: VAXserver 3800 under VAX/VMS Version 5.2

Target: Motorola MVME 133A-20 (MC68020) (bare machine)

Communication Network: RS 232

A tape containing the customized test suite was loaded onto the host computer. Results were collected on the host computer and transferred via Ethernet to another computer for evaluation and archiving.

The compiler was tested using command scripts provided by TeleSoft and reviewed by the validation team. The tests were compiled using the command

```
TSADA/E68/ADA/OPTIMIZE/VIRTUAL_SPACE=3000/-
    ENABLE/CPU=MC68020/FP_INLINE  <test file>
```

and linked with the command

```
TSADA/E68/LINK/LOAD_MODULE=<testname>/-
    OPTIONS=<options file> <main unit>
```

The qualifiers /LIST and /SWITCH="+LIST_GEN" were added to the compiler call for class B, expanded and modified tests. See Appendix E for explanation of compiler and linker switches. The <options file> contained a specification of memory addresses for the target computer.

Tests were compiled, linked, and executed (as appropriate) using two identical host and two identical target computers. Test output, compilation listings, and job logs were captured on cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at TeleSoft, San Diego, USA, and was completed on 31 January 1990.

APPENDIX A

DECLARATION OF CONFORMANCE

TeleSoft has submitted the following Declaration of Conformance concerning the TeleGen2 Ada Development System.

DECLARATION OF CONFORMANCE

Compiler Implementor: TELESOFT
Ada Validation Facility: IABG, Dept. SZT. D-8012 Ottobrunn
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	TeleGen2 Ada Development System for VAX to E68K, Version 4.0
Version:	4.0
Host Computer System:	VAXserver 3800 (under VAX/VMS Version 5.2)
Target Computer System:	Motorola MVME 133A-20 (MC68020) (Bare machine)

Customer's Declaration

I, the undersigned, representing TELESOFT, declare that TELESOFT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.

Date: 2-1-90

TELESOFT
Raymond A. Parra, Vice President and General Counsel

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the TeleGen2 Ada Development System, as described in this Appendix, are provided by TeleSoft. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

      type SHORT_INTEGER is range -128 .. 127;
      type INTEGER is range -32768 .. 32767;
      type LONG_INTEGER is range -2147483648 .. 2147483647;

      type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
      type LONG_FLOAT is digits 15
         range -8.98846567431158E+307 .. 8.98846567431158E+307;

      type DURATION is delta 2#1.0#E-14 range -86400.0 .. 86400.0;

   ...

end STANDARD;
```

CHAPTER 3: LRM ANNOTATIONS

CHAPTER CONTENTS

CHAPTER 3: LRM ANNOTATIONS

CHAPTER CONTENTS

3. LRM ANNOTATIONS

TeleGen2 compiles the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM) (ANSI/MIL-STD-1815A). This chapter describes the portions of the language that are designated by the LRM as implementation dependent for the compiler and run-time environment.

The information is presented in the order in which it appears in the LRM. In general, however, only those language features that are not fully implemented by the current release of TeleGen2 or that require clarification are included. The features that are optional or that are implementation dependent, on the other hand, are described in detail. Particularly relevant are the sections annotating LRM Chapter 13 (Representation Clauses and Implementation-Dependent Features) and Appendix F (Implementation-Dependent Characteristics).

3.1. LRM Chapter 2 - Lexical Elements

[LRM 2.1] Character Set. The host and target character set is the ASCII character set.

[LRM 2.2] Lexical Elements, Separators, and Delimiters. The maximum number of characters on an Ada source line is 200.

[LRM 2.8] Pragmas. TeleGen2 implements all language-defined pragmas *except* pragma Optimize. If pragma Optimize is included in Ada source, the pragma will have no effect. Optimization is implemented by using pragma Inline and the optimizer. Pragma Inline is not supported for library-level subprograms.

Limited support is available for pragmas Memory_Size, Storage_Unit, and System_Name; that is, these pragmas are allowed if the argument is the same as the value specified in the System package.

Pragmas Page and List are supported in the context of source/error listings; refer to the Compiler/Linker chapter of the TeleGen2 *User Guide* for more information.

3.2. LRM Chapter 3 - Declarations and Types

[LRM 3.2.1] Object Declarations. TeleGen2 does not produce warning messages about the use of uninitialized variables. The compiler will not reject a program merely because it contains such variables.

[LRM 3.5.1] Enumeration Types. The maximum number of elements in an enumeration type is 32767. This maximum can be realized only if generation of the image table for the type has been deferred, and there are no references in the program that would cause the image table to be generated. Deferral of image table generation for an enumeration type, P, is requested by the statement:

```
pragma Images (P, Deferred);
```

Refer to "Implementation-Defined Pragmas," in Section 3.8.1, for more information on pragma Images.

[LRM 3.5.4] Integer Types. There are three predefined integer types: Short_Integer, Integer, and Long_Integer. The attributes of these types are shown in Table 3-1. Note that using explicit integer type definitions instead of predefined integer types should result in more portable code.

Table 3-1. Attributes of Predefined Types Short_Integer, Integer, and Long_Inte

Attribute	Type		
	Short_Integer	Integer	Long_Integer
'First	-128	-32768	-2147483648
'Last	127	32767	2147483647
'Size	8	16	32
'Width	4	6	11

[LRM 3.5.8] Operations of Floating Point Types. There are two predefined floating point types: Float and Long_Float. The attributes of types Float and Long_Float are shown in Table 3-2. This floating point facility is based on the IEEE standard for 32-bit and 64-bit numbers. Note that using explicit real type definitions should lead to more portable code.

The type Short_Float is not implemented.

Table 3-2. Attributes of Predefined Types Float and Long_Float

Attribute	Type	
	Float	Long_Float
'Machine_Overflows	TRUE	TRUE
'Machine_Rounds	TRUE	TRUE
'Machine_Radix	2	2
'Machine_Mantissa	24	53
'Machine_Emax	127	1023
'Machine_Emin	-125	-1021
'Mantissa	21	51
'Digits	6	15
'Size	32	64
'Emax	84	204
'Safe_Emax	125	1021
'Epsilon	9.53674E-07	8.88178E-16
'Safe_Large	4.25253E+37	2.24711641857789E+307
'Safe_Small	1.17549E-38	2.22507385850721E-308
'Large	1.93428E+25	2.57110087081438E+61
'Small	2.58494E-26	1.99469227433161E-62

3.3. LRM Chapter 4 - Names and Expressions

[LRM 4.10] Universal Expressions. There is no limit on the accuracy of real literal expressions. Real literal expressions are computed using an arbitrary-precision arithmetic package.

3.4. LRM Chapter 9 - Tasks

[LRM 9.6] Delay Statements, Duration, and Time. This implementation uses 32-bit fixed point numbers to represent the type Duration. The attributes of the type Duration are shown in Table 3-3.

Table 3-3. Attributes of Type Duration

Attribute	Value
'Delta	0
'First	-86400
'Last	86400

[LRM 9.8] Priorities. Sixty-four levels of priority are available to associate with tasks through pragma Priority. The predefined subtype Priority is specified in the package System as

```
subtype Priority is Integer range 0..63;
```

Currently the priority assigned to tasks without a pragma Priority specification is 31; that is:

```
(System.Priority'First + System.Priority'Last) / 2
```

[LRM 9.11] Shared Variables. The restrictions on shared variables are only those specified in the LRM.

3.5. LRM Chapter 10 - Program Structure and Compilation Issues

[LRM 10.1] Compilation Units - Library Units. All main programs are assumed to be parameterless procedures or functions that return an integer result type.

3.6. LRM Chapter 11 - Exceptions

[LRM 11.1] Exception Declarations. Numeric_Error is raised for integer or floating point overflow and for divide-by-zero situations. Floating point underflow yields a result of zero without raising an exception.

Program_Error and Storage_Error are raised by those situations specified in LRM Section 11.1. Exception handling is also discussed in the Programming Guide chapter.

3.7. LRM Chapter 13 - Implementation-Dependent Features

As shown in Table 3-4, the current release of TeleGen2 supports most LRM Chapter 13 facilities. The sections below the table document those LRM Chapter 13 facilities that are either not implemented or that require explanation. Facilities implemented exactly as described in the LRM are not mentioned.

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2

13.1 Representation Clauses	Supported, except as indicated below (LRM 13.2 – 13.5). Pragma Pack is supported, *except for* dynamically sized components. For details on the TeleGen2 implementation of pragma Pack, see Section 3.7.1.
13.2 Length Clauses	Supported: 'Size 'Storage_Size for collections 'Storage_Size for task activation 'Small for fixed-point types See Section 3.7.2 for more information.
13.3 Enumeration Rep. Clauses	Supported, *except for* type Boolean or types derived from Boolean. (Note: users can easily define a non-Boolean enumeration type and assign a representation clause to it.)
13.4 Record Rep. Clauses	Supported *except for* records with dynamically sized components. See Section 3.7.4 for a full discussion of the TeleGen2 implementation.
13.5 Address Clauses	*Supported for:* objects (including task objects). *Not supported for:* packages, subprograms, or task units. See Section 3.7.5 for more information.
13.5.1 Interrupts	For interrupt entries, the address of a TeleGen2-defined *interrupt descriptor* can be given. See "Interrupt Handling" in the Programming Guide chapter for more information.
13.6 Change of Representation	Supported, *except for* types with record representation clauses.
------ *Continued on the next page* -----	

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2 (Contd)

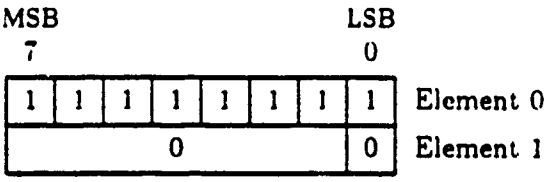
------ *Continued from the previous page* -----		
13.7	Package System	Conforms closely to LRM model. Refer to Section 3.7.7 for details on the TeleGen2 implementation.
13.7.1	System-Dependent Named Numbers	Refer to the specification of package System (Section 3.7.7).
13.7.2	Representation Attributes	Implemented as described in LRM *except that*: 'Address for packages is unsupported. 'Address of a constant yields a null address.
13.7.3	Representation Attributes of Real Types	See Table 3-2.
13.8	Machine Code Insertions	Fully supported. The TeleGen2 implementation defines an attribute, 'Offset, that, along with the language-defined attribute 'Address, allows addresses of objects and offsets of data items to be specified in stack frames. Refer to "Using Machine Code Insertions" in the Programming Guide chapter for a full description on the implementation and use of machine code insertions.
13.9	Interface to Other Languages	Pragma Interface is supported for Assembly, C, UNIX, and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for a description of the implementation and use of pragma Interface.
13.10	Unchecked Programming	Supported except as noted below (LRM 13.10.2).
13.10.1	Unchecked Storage Deallocation	Supported.
13.10.2	Unchecked Type Conversions	Supported *except for* unconstrained record or array types.

3.7.1. Pragma Pack. This section discusses how pragma Pack is used in the TeleGen2 implementation.

a. With Boolean Arrays. You may pack Boolean arrays by the use of pragma Pack. The compiler allocates 8 bits for a single Boolean, 8 bits for a component of an unpacked Boolean array, and 1 bit for a component of a packed Boolean array. The first figure illustrates the layout of an unpacked Boolean array; the one below that illustrates a packed Boolean array:

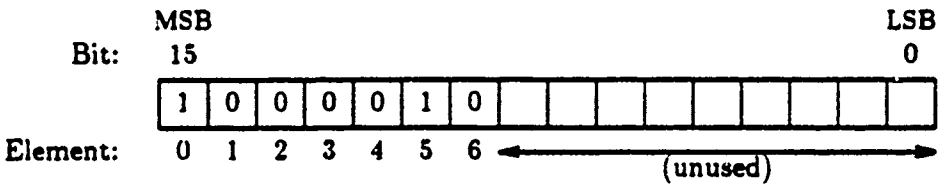
---------- Unpacked Boolean array: --------

```
Unpacked_Bool_Arr_Type is array (Natural range 0..1) of Boolean
U_B_Arr: Unpacked_Bool_Arr_Type := (True,False);
```



---------- Packed Boolean array: --------

```
Packed_Bool_Arr_Type is array (Natural range 0..6) of Boolean;
    pragma Pack (Packed_Bool_Arr_Type);
    P_B_Arr: Packed_Bool_Arr_Type := (P_B_Arr(0) => True,
      P_B_Arr(5) => True, others => False);
```



b. With Records. You may pack records by use of pragma Pack. Packed records follow these conventions:

1. The total size of the record is a multiple of 8 bits.
2. Packed records may cross word boundaries.
3. Records are packed to the bit level if the elements are themselves packed.

Below is an example of packing in a procedure, Rep_Proc, that defines three records of different lengths. Objects of these three packed record types are components of the packed record Rec. The storage allocated for Rec is 16 bits; that is, it is maximally packed.

```
procedure Rep_Proc is

  type A1 is array (Natural range 0 .. 8) of Boolean;
  pragma Pack (A1);

  type A2 is array (Natural range 0 .. 3) of Boolean;
  pragma Pack (A2);

  type A3 is array (Natural range 0 .. 2) of Boolean;
  pragma Pack (A3);

  type A_Rec is
    record
      One   : A1;
      Two   : A2;
      Three : A3;
    end record;
  pragma Pack (A_Rec);

  Rec   : A_Rec;

begin
  Rec.One       := ( 0 => True,   1 => False,  2 => False,
                     3 => False,  4 => True,   5 => False,
                     6 => False,  7 => False,  8 => True );
  Rec.Two (3)   := True;
  Rec.Three (1) := True;
end Rep_Proc;
```

3.7.2. [LRM 13.2] Length Clauses. A length clause specifies an amount of storage associated with a type. The sections below describe how length clauses are supported in this implementation of TeleGen2 and how to use length clauses effectively within the context of TeleGen2.

3.7.2.1. (a) Specifying Size: T'Size. The prefix T denotes an object. The size specification must allow for enough storage space to accommodate every allowable value of these objects. The constraints on the object and on its subcomponents (if any) must be static. For an unconstrained array type, the index subtypes must also be static.

For this implementation, Min_Size is the smallest number of bits logically required to hold any value in the range; no sign bit is allocated for non-negative ranges. Biased representations are not supported; e.g., a range of 100 .. 101 requires 7 bits, not 1. Warning: in the current release, using a size clause for a discrete type may cause inefficient code to be generated. For example, given...

```
type Nibble is range 0 .. 15;
for Nibble'Size use 4;
```

...each object of type Nibble will occupy only 4 bits, and relatively expensive bit-field instructions will be used for operations on Nibbles. (A single declared object of type Nibble will be aligned on a storage-unit boundary, however.)

For floating-point and access types, a size clause has no effect on the representation. (Task types are implemented as access types).

For composite (array or record) types, a size clause acts like an implicit pragma Pack, followed by a check that the resulting size is no greater than the requested size. Note that the composite type will be packed whether or not it is necessary to meet the requested size. The size clause for a record must be a multiple of storage units.

3.7.2.2. (b) Specifying Collection Size: T'Storage_Size. A collection is the entire set of objects created by evaluation of allocators for an access type.

The prefix T denotes an access type. Given an access type Acc_Type, a length clause for a collection allocated using Acc_Type objects might look like this:

```
for Acc_Type'Storage_Size use 64;
```

In TeleGen2, the above length clause allocates from the heap 64 bytes of contiguous memory for objects created by Acc_Type allocators. Every time a new object is created, it is put into the remaining free part of the memory allocated for the collection, provided there is adequate space remaining in the collection. Otherwise, a storage error is raised.

Keeping the objects in a contiguous span of memory allows system storage reclamation routines to deallocate and manage the space when it is no longer needed. Pragma Controlled can prevent the deallocation of a specified collection of objects. Objects can be explicitly deallocated by calling the Unchecked_Deallocation procedure instantiated for the object and access types.

Given an access type which does not have a length clause specified, the 'Storage_Size attribute will return a value of 0.

Header Record

In this configuration of TeleGen2, information needed to manage storage blocks in a collection is stored in a collection header that requires 20 bytes of memory, adjacent to the collection, in addition to the value specified in the length clause.

Minimum Size

When an object is deallocated from a collection, a record containing link and size information for the space is put in the deallocated space as a placeholder. This enables the space to be located and reallocated. The space allocated for an object must therefore have the minimum size needed for the placeholder record. For this TeleGen2 configuration, this minimum size is the sum of the sizes of an access type and a integer type, or 6 bytes.

Dynamically Sized Objects

When a dynamically-sized object is allocated, a record requiring 2 bytes accompanies it to keep track of the size of the object for when it is put on the free list. The record is used to set the size field in the placeholder record since compaction may modify the value.

Size Expressions

Instead of specifying an integer in the length clause, you can use an expression to specify storage for a given number of objects. For example, suppose an access type Dict_Ref references a record Symbol_Rec containing five fields:

```
type Tag is String(1..8);

type Symbol_Rec;
type Dict_Ref is access Symbol_Rec;

type Symbol_Rec is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : Tag;
   end record;
```

To allocate 10 Symbol_Rec objects, you could use an expression such as:

```
for Dict_Ref'Storage_Size use ((Symbol_Rec'Size * 10)+20);
```

where 20 is the extra space needed for the header record. (Symbol_Rec is obviously larger than the minimum size required, which is equivalent to one access type and one integer.)

In another implementation, Symbol_Rec might be a variant record that uses a variable length for the string Key:

```
type Symbol_Rec(Last : Natural :=0) is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : String(1..Last);
   end record;
```

In this case, Symbol_Rec objects would be dynamically sized depending on the length of the string for Key. Using a length clause for Dict_Ref as above would then be illegal since Symbol_Rec'Size cannot be consistently determined. A length clause for Symbol_Rec objects, as described in (a) above, would be illegal since not all components of Symbol_Rec are static. As defined, a Symbol_Rec object could conceivably have a Key string with Integer'Last number of characters.

3.7.2.3. (c) Specifying Storage for Task Activation: T'Storage_Size. The prefix T denotes a task type. A length clause for a task type specifies the number of storage units to be reserved for an activation of a task of the type. The TeleGen2 default stack size is 4000 bytes.

3.7.2.4. (d) Specifying 'Small for Fixed Point Types: T'Small. Small is the absolute precision (a positive real number) while the prefix T denotes the first named subtype of a fixed point type. Elaboration of a real type defines a set of model numbers. T'Small is generally a power of 2, and model numbers are generally multiples of this number so that they can be represented exactly on a binary machine. All other real values are defined in terms of model numbers having explicit error bounds.

Example:

```
type Fixed is delta 0.25 range -10.0 .. 10.0;
```

Here...

```
Fixed'Small = 0.25    -- A power of 2

3.0 = 12 * 0.25       -- A model number but not a power of 2
```

The value of the expression of the length clause must not be greater than the delta of the first named subtype. The effect of the length clause is to use this value of 'Small for the representation of values of the fixed point base type. The length clause thereby also affects the amount of storage for objects that have this type.

If a length clause is not used, for model numbers defined by a fixed point constraint, the value of Small is defined as the largest power of two that is not greater than the delta of the fixed accuracy definition.

If a length clause is used, the model numbers are multiples of the specified value for Small. For this configuration of TeleGen2, the specified value must be (mathematically) equal to either an exact integer or the reciprocal of an exact integer.

Examples:

```
1.0, 2.0, 3.0, 4.0, . . . are legal
0.5, 1.0/3.0, 0.25, 1.0/3600.0 are legal
2.5, 2.0/3.0, 0.3 are illegal
```

3.7.3. [LRM 13.3] Enumeration Representation Clauses. Enumeration representation clauses are supported, except for Boolean types.

Performance note: Be aware that use of such clauses will introduce considerable overhead into many operations that involve the associated type. Such operations include indexing an array by an element of the type, or computing the 'Pos, 'Pred, or 'Succ attributes for values of the type.

3.7.4. [LRM 13.4] Record Representation Clauses. Since record components are subject to rearrangement by the compiler, you must use representation clauses to guarantee a particular layout. Such clauses are subject to the following constraints:

- Each component of the record must be specified with a component clause.
- The alignment of the record is restricted to mods 1, 2, and 4; byte, word, and long-word aligned, respectively.
- Bits are ordered right to left within a byte.
- Components may cross word boundaries.

Here is a simple example showing how the layout of a record can be specified by using representation clauses:

```
package Repspec_Example is
  Bits : constant := 1;
  Word : constant := 4;

  type Five is range 0 .. 16#1F#;
  type Seventeen is range 0 .. 16#1FFFF#;
  type Nine is range 0 .. 511;

  type Record_Layout_Type is record
```

```
    Element1 : Seventeen;
    Element2 : Five;
    Element3 : Boolean;
    Element4 : Nine;
  end record;

for Record_Layout_Type use record at mod 2;
    Element1 at 0*Word range 0 .. 16;
    Element2 at 0*Word range 17 .. 21;
    Element3 at 0*Word range 22 .. 22;
    Element4 at 0*Word range 23 .. 31;
  end record;

Record_Layout : Record_Layout_Type;
end Repspec_Example;
```

3.7.5. [LRM 13.5] Address Clauses. The Ada compiler supports address clauses for objects and entries. Address clauses for packages and task units are not supported.

Address clauses for objects may be used to access hardware memory registers or other known memory locations. The use of address clauses is affected by the fact that the System.Address type is private. For the MC680x0 target, literal addresses are represented as integers, so an unchecked conversion must be applied to these literals before they can be passed as parameters of type System.Address. For example, in the examples in this document the following declaration is often assumed:

```
function Addr is new Unchecked_Conversion (Long_Integer,System.Address);
```

This function is invoked when an address literal needs to be converted to an Address type. Naturally, user programs may implement a different convention. Below is a sample program that uses address clauses and this convention. Package System must be explicitly *with*ed when using address clauses.

```
with System;
with Unchecked_Conversion;
procedure Hardware_Access is
  function Addr is new Unchecked_Conversion (Long_Integer, System.Address);
  Hardware_Register : integer;
  for Hardware_Register use at Addr (16#FF0000#);
begin
  ...
end Hardware_Access;
```

When using an address clause for an object with an initial value, the address clause should immediately follow the object declaration:

```
Obj: Some_Type := <init_expr>;
for Obj use at <addr_expr>;
```

This sequence allows the compiler to perform an optimization wherein it generates code to evaluate the <addr_expr> as part of the elaboration of the declaration of the object. The expression <init_expr> will then be evaluated and assigned directly to the object, which is stored at <addr_expr>. If another declaration had intervened between the object declaration and the address clause, the compiler would have had to create a temporary object to hold the initialization value before copying it into the object when the address clause is elaborated. If the

object were a large composite type, the need to use a temporary could result in considerable overhead in both time and space. To optimize your applications, therefore, you are encouraged to place address clauses immediately after the relevant object declaration.

As mentioned above, arrays containing components that can be allocated in a signed or unsigned byte (8 bits) are packed, one component per byte. Furthermore, such components are referenced in generated code by MC680x0 byte instructions. The following example indicates how these facts allow access to hardware byte registers:

```
with System;
with Unchecked_Conversion;
procedure Main is
  function Addr is new Unchecked_Conversion (Long_Integer, System.Address);
  type Byte is range -128..127;
  HW_Regs : array (0..1) of Byte;
  for HW_Regs use at Addr (16#FFF310#);

  Status_Byte : constant integer := 0;
  Next_Block_Request: constant integer := 1;
  Request_Byte : Byte := 119;
  Status : Byte;

begin
  Status := HW_Regs(Status_Byte);
  HW_Regs(Next_Block_Request) := Request_Byte;
end Main;
```

Two byte hardware registers are referenced in the example above. The status byte is at location 16#FFF310# and the next block request byte is at location 16#FFF311#.

Function Addr takes a long integer as its argument. Long_Integer'Last is 16#7FFFFFFF#, but there are certainly addresses greater than Long_Integer'Last. Those addresses with the high bit set, such as FFFA0000, cannot be represented as a positive long integer. Thus, for addresses with the high bit set, the address should be computed as the negation of the 2's complement of the desired address. According to this method, the correct representation of the sample address above would be Addr(-16#00060000#).

3.7.6. [LRM 13.6] Change of Representation. TeleGen2 supports changes of representation, except for types with record representation clauses.

3.7.7. [LRM 13.7] The Package System. The specification of TeleGen2's implementation of package System is presented in the LRM Appendix F section at the end of this chapter.

3.7.8. [LRM 13.7.2] Representation Attributes. The compiler does not support 'Address for packages.

3.7.9. [LRM 13.7.3] Representation Attributes of Real Types. The representation attributes for the predefined floating point types were presented in Table 3-2.

3.7.10. [LRM 13.8] Machine Code Insertions. Machine code insertions, an optional feature of the Ada language, are fully supported in TeleGen2. Refer to the "Using Machine Code Insertions" section in the Programming Guide chapter for information regarding their

implementation and for examples on their use.

3.7.11. [LRM 13.9] Interface to Other Languages. In TeleGen2, pragma Interface is supported for Assembly, C, UNIX, and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for for information on the use of pragma Interface. TeleGen2 does not currently allow pragma Interface for library units.

3.7.12. [LRM 13.10] Unchecked Programming. Unchecked_Conversion is allowed except when the target data subtype is an unconstrained array or record type. If the size of the source and target are static and equal, the compiler will perform a bitwise copy of data from the source object to the target object.

Where the sizes of source and target differ, the following rules will apply.

- If the size of the source is greater than the size of the target, the high address bits will be truncated in the conversion.
- If the size of the source is less than the size of the target, the source will be moved into the low address bits of the target.

The compiler will issue a warning when Unchecked_Conversion is instantiated with unequal sizes for source and target subtype. Unchecked_Conversion between objects of different or non-static sizes will usually produce less efficient code and should be avoided, if possible.

3.8. LRM Appendix F for TeleGen2

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. Table 3-5 constitutes Appendix F for this implementation. Points that require further clarification are addressed in sections referenced in the table.

Table 3-5. LRM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All (Section 3.8.1). (b) Predefined pragmas with implementation-dependent characteristics: * Interface (assembly, UNIX, C, and Fortran—see "Interfacing to Other Languages." Not supported for library units. * List and Page (in context of source/error compiler listings.) (See the *User Guide.*) * Pack. See Section 3.7.1. * Inline. Not supported for library-level subprograms. * Priority. Not supported for main programs. *Other supported predefined pragmas:* Controlled Shared Suppress Elaborate *Predefined pragmas partly supported* (see Section 3.1): Memory_Size Storage_Unit System_Name *Not supported:* Optimize
(2) Implementation-Dependent Attributes	'Offset. Used for machine code insertions. The predefined attribute 'Address is not supported for packages. See "Using Machine Code Insertions" earlier in this chapter for information on 'Offset and 'Address. 'Extended_Image 'Extended_Value 'Extended_Width 'Extended_Aft 'Extended_Digits Refer to Section 3.8.2 for information on the implementation-defined extended attributes listed above.
(3) Package System	See Section 3.7.7.
(4) Restrictions on Representation Clauses	Summarized in Table 3-4.
------- *Continued on the next page* -------	

Table 3-5. LRM Appendix F for TeleGen2 (Contd)

------- *Continued from the previous page* -------	
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Summarized in Table 3-4.
(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as follows: type Count is range 0..(2 ** 31)-2 2. In Text_IO, the type Field is defined as follows: subtype Field is integer range 0..1000 3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.) 4. Sequential_IO and Direct_IO can be instantiated for unconstrained array types or discriminated types without defaults. 5. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Integer and Long_Integer and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages: Short_Integer_Text_IO Integer_Text_IO Long_Integer_Text_IO Float_Text_IO Long_Float_Text_IO

3.8.1. Implementation-Defined Pragmas. There are eight implementation-defined pragmas in TeleGen2: pragmas Comment, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All.

3.8.1.1. Pragma Comment. Pragma Comment is used for embedding a comment into the object code. Its syntax is:

```
pragma Comment ( <string_literal> );
```

where "<string_literal>" represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

3.8.1.2. Pragma Images. Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type; the length of the index table is one greater than the number of literals.

The syntax of this pragma is:

pragma Images(<enumeration_type>, Deferred);
-- or --
pragma Images(<enumeration_type>, Immediate);

The default, Deferred, saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, using

pragma Images(<enumeration_type>, Immediate);

will cause a single image table to be generated in the literal pool of the unit declaring the enumeration type.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

pragma Images(<enumeration_type>, Immediate);

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler.

3.8.1.3. Pragma Interface_Information. The existing Ada interface pragma only allows specification of a language name. In some cases, the optimizing code generator will need more information than can be derived from the language name. Therefore there is a need for an implementation-specific pragma, Interface_Information.

There is an extended usage of this pragma for Machine Code Insertion procedures which does not use a preceding pragma Interface. Other than that case, a pragma Interface_Information is always associated with a Pragma Interface. The syntax is:

```
pragma Interface_Information (Name,
                              Link_Name,
                              Mechanism,
                              Parameters,
                              Clobbered_Regs);
```

where:

```
name                :== ada_subprogram_identifier, required
link_name           :== string, default = ""
mechanism           :== string, default = "PROTECTED"
parameters          :== string, default = ""
clobbered_regs      :== string, default = ""
```

Scope.

Pragma Interface_Information is allowed wherever the standard pragma Interface is allowed, and must be immediately preceded by a pragma Interface purporting to the same Ada subprogram. in the same declarative part or package specification; no intervening declaration is allowed between the Interface and Interface_Information pragmas. Contrary to pragma Interface, this pragma is not allowed for overloaded subprograms (it specifies information that pertain to one specific body of non-Ada code). If the user wishes to use overloaded Ada names, the Interface and Interface_Information pragmas may be applied to unique renaming declarations.

The pragma is also allowed for a library unit; in that case, the pragma must occur immediately after the corresponding Interface pragma, and before any subsequent compilation unit.

This pragma may be applied to any interfaced subprogram. regardless of the language or system named in the interface pragma. The code generator is responsible for rejecting or ignoring illegal or redundant interface information. The optimizing code generator will process and check the legality of such interfaced subprograms at the time of the spec compilation, instead of waiting for an actual use of the interfaced subprogram. This will save the user from extensive recompilation of the offensive specification and all its dependents should an illegal pragma have been used.

This pragma is also used for Machine Code Insertion (MCI) procedures. In that case. the "mechanism" should be set to "mci." This allows the user to specify detailed parameter characteristics for the call and inlined call to the MCI procedure. When used in conjunction with pragma Inline. this allows the user to directly insert a minimal set of instructions into the call location.

Parameters.

Name: Ada subprogram identifier. The rule detailed in LRM 13.9 for a subprogram named in a pragma Interface apply here as well. As explained above, the subprogram must have been named in an immediately preceding Interface pragma.

This is the only required parameter. Since the other parameters are optional, positional association may only be used if all parameters are specified, or only the rightmost ones are defaulted. Named association must be used otherwise: this requires that the front-end supports it in pragmas.

Link_Name: string literal. When specified, this parameter indicates the name the code generator must use to reference the named subprogram. This string name may contain any characters allowed in an Ada string and must be passed unchanged (in particular, not case-mapped) to the code generator. The code generator will reject names that are illegal in the particular language or system being targeted.

If this parameter is not specified, it defaults to a null string. The code generator will interpret a default link_name differently, depending on the target language/system (the default is generally the Ada name, or is derived from it, for example, "_Ada_name" for 'C' calls).

Mechanism: string literal. The only mechanism currently implemented is the "mci" mechanism used strictly in conjunction with Machine Code Insertion procedures.

A future mechanism will include "protected" where the code generator will protect the Ada exception model on calls to that external subprogram. Unfortunately, such protection is costly, which means that it should be applied only when necessary, as most interfaced subprograms do not raise exceptions or traps. Until this mechanism is implemented, external code which causes traps will cause unpredictable unhandled exception tracebacks.

Parameters: string literal. This string, when present, tells the code generator that some parameters are to be passed in registers (instead of on the stack, which is the default). The string is passed as is by the front-end, and is decoded by the code generator.

The code generator interprets the string as a positional aggregate, where each position refers to a parameter of the interfaced subprogram. Each position of the aggregate may either be null, or one of the following identifiers: "D0, D1, A0, A1, FP0, or FP1," which specify that the corresponding parameter be passed in the named register. Thus the string "A0,,D0" specifies that the first parameter be passed in A0, the second in the stack, the third in D0, and any other parameters in the stack.

A register identifier may be used, at most, once in the parameter passing description string, except for function return value, as explained below. The code generator will also enforce some semantic restrictions; for example, a floating point register can only be used to pass a float type parameter, although a float parameter can be passed in a regular data register. Only scalar values and addresses may be passed directly in registers; composites must be passed in the stack (by reference or by value, dictated by the code generator model for the target language/system). Explicit passing of the 'Address of a composite in a register is allowed (the parameter must be of type System.Address).

The parameters string can also be used to specify the register in which a function return value will be passed back. The parameters registers string may in that case have one position more than the number of parameters; this position describes the return value. The register used for the return value may have been used for an input parameter. This is the only case where a register may occur twice in the same string. If not specified for a given function, the code generator uses the register normally used for the target language/system (that is, "C" returns all in D0 or D0/D1, whereas "Assembly" returns in D0, D0/D1, A0, or FP0 depending on the return value type).

Clobbered_Regs. These are the registers (comma-separated list) that are destroyed by this operation. The code generator will keep anything valuable in these registers at the point of the call.

A simple example of the use of pragma Interface_Information is:

```
procedure Do_Something (Addr: System.Address; Len: Integer);
pragma Interface (Assembly, Do_Something);
pragma Interface_Information ( Name       => Do_Something,
                               Link_Name  => "CGS$$DOIT",
                               Mechanism  => "UNPROTECTED",
                               Parameters => "AO,DO");
```

3.8.1.4. Pragma Interrupt. Pragma Interrupt is described in this reference volume in Chapter 2. Please refer to the section "Optimized Interrupt Entries," subsection "Function-Mapped Optimizations."

3.8.1.5. Pragma Linkname. Pragma Linkname is used to provide interface to any routine whose name can be specified by an Ada string literal. This allows access to routines whose identifiers do not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is:

pragma Interface (assembly, <subprogram_name>);
pragma Linkname (<subprogram_name>, <string_literal>);

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect.

A simple example of the use of pragma Linkname is:

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

Note: It is preferable that the user use pragma Interface_Information for this functionality.

3.8.1.6. Pragma No_Suppress. No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is:

pragma No_Suppress (<identifier> [, [ON =>] <name>]);

where **<identifier>** is the type of check you want to suppress (e.g., access_check; refer to LRM 11.7)

<name> is the name of the object, type/subtype. task unit, generic unit, or subprogram within which the check is to be suppressed; <name> is optional.

If neither Suppress nor No_Suppress is present in a program, checks will not be suppressed. You may override this default at the command level, by compiling the file with the -i(nhibit option and specifying with that option the type of checks you want to suppress. For more information on -i(nhibit, refer to your TeleGen2 *Overview and Command Summary* document.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over an -i(nhibit option provided during compilation.

3.8.1.7. Pragma Preserve_Layout. The TeleGen2 compiler reorders record components to minimize gaps within records. Pragma Preserve_Layout forces the compiler to maintain the Ada source order of components of a given record type, thereby preventing the compiler from performing this record layout optimization.

The syntax of this pragma is:

Pragma Preserve_Layout (ON => Record_Type_Name)

Preserve_Layout must appear before any forcing occurrences of the record type and must be in the same declarative part, package specification, or task specification. This pragma can be applied to a record type that has been packed. If Preserve_Layout is applied to a record type that has a record representation clause, the pragma only applies to the components that do not have component clauses. These components will appear in Ada source order after the components with component clauses.

3.8.1.8. Pragma Suppress_All. Suppress_All is a TeleGen2-defined pragma that will suppress all checks in a given scope. Pragma Suppress_All contains no arguments and can be placed in the same scopes as pragma Suppress.

In the absence of pragma Suppress_All or any other suppress pragma, the scope which contains the pragma will have checking turned off. This pragma should be used in a safe piece of time critical code to allow for better performance.

3.8.2. Implementation-Dependent Attributes.

3.8.2.1. 'Address and 'Offset. These were discussed within the context of using machine code insertions, in the Programming Guide chapter.

3.8.2.2. Extended Attributes for Scalar Types. The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.
2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

Integer:	'Extended_Image,	'Extended_Value,	'Extended_Width
Enumeration:	'Extended_Image,	'Extended_Value,	'Extended_Width
Floating Point:	'Extended_Image,	'Extended_Value,	'Extended_Digits
Fixed Point:	'Extended_Image, 'Extended_Aft	'Extended_Value,	'Extended_Fore,

The extended attributes can be used without the overhead of including Text_IO in the linked program. Below is an example that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is
Temp_Str : String ( 1 .. 6 + Float'Digits );
package Flt_IO is new Text_IO.Float_IO (Float);
begin
  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;
end Convert_To_String;

    function Convert_To_String_No_Text_IO( Fl : Float ) return String is
    begin
      return Float'Extended_Image ( Fl );
    end Convert_To_String_No_Text_IO;


with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is
Value : Float := 10.03376;
begin
  Text_IO.Put_Line ( "Using the Convert_To_String, the value of the variable
  is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_IO, the value
  is : " & Convert_To_String_No_Text_IO ( Value ) );
end Show_Different_Conversions;
```

3.8.2.2.1. Integer Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Base,Based,Space_IF_Positive)

Returns the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared:

```
subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Image(5)                    = "5"
X'Extended_Image(5,0)                  = "5"
X'Extended_Image(5,2)                  = " 5"
X'Extended_Image(5,0,2)                = "101"
X'Extended_Image(5,4,2)                = " 101"
X'Extended_Image(5,0,2,True)           = "2#101#"
X'Extended_Image(5,0,10,False)         = "5"
X'Extended_Image(5,0,10,False,True)    = " 5"
X'Extended_Image(-1,0,10,False,False)  = "-1"
X'Extended_Image(-1,0,10,False,True)   = "-1"
X'Extended_Image(-1,1,10,False,True)   = "-1"
```

```
X'Extended_Image(-1,0,2,True,True)      = "-2#1#"
X'Extended_Image(-1,10,2,True,True)     = "      -2#1#"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Description:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. *Required*

Examples:

Suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Value("5")        = 5
X'Extended_Value(" 5")       = 5
X'Extended_Value("2#101#")   = 5
X'Extended_Value("-1")       = -1
X'Extended_Value(" -1")      = -1
```

'Extended_Width

Usage:

X'Extended_Width(Base,Based,Space_If_Positive)

Returns the width for subtype of X.

For a prefix X that is a discrete subtype: this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameter Descriptions:

Base	The base for which the width will be calculated. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Width                    = 3  -- "-10"
X'Extended_Width(10)                = 3  -- "-10"
X'Extended_Width(2)                 = 5  -- "10000"
X'Extended_Width(10,True)           = 7  -- "-10#10#"
X'Extended_Width(2,True)            = 8  -- "2#10000#"
X'Extended_Width(10,False,True)     = 3  -- " 16"
X'Extended_Width(10,True,False)     = 7  -- "-10#10#"
X'Extended_Width(10,True,True)      = 7  -- " 10#16#"
X'Extended_Width(2,True,True)       = 9  -- " 2#10000#"
X'Extended_Width(2,False,True)      = 6  -- " 10000"
```

3.8.2.2.2. Enumeration Type Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Uppercase)

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. If the Width specified is larger than the image of Item, the return string is padded with trailing spaces. If the Width specified is smaller than the image of Item, the default is assumed and the image of the enumeration value is output completely. *Optional*
Uppercase	An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. *Optional*

Examples:

Suppose the following types were declared:

```
type X is (red, green, blue, purple);
type Y is ('a', 'B', 'c', 'D');
```

Then the following would be true:

```
X'Extended_Image(red)              = "RED"
X'Extended_Image(red, 4)           = "RED "
X'Extended_Image(red,2)            = "RED"
X'Extended_Image(red,0,false)      = "red"
X'Extended_Image(red,10,false)     = "red       "
Y'Extended_Image('a')              = "'a'"
Y'Extended_Image('B')              = "'B'"
Y'Extended_Image('a',6)            = "'a'  "
Y'Extended_Image('a',0,true)       = "'a'"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. *Required*

Examples:

Suppose the following type were declared:

```
type X is (red, green, blue, purple);
```

Then the following would be true:

```
X'Extended_Value("red")          = red
X'Extended_Value(" green")       = green
X'Extended_Value("    Purple")   = purple
X'Extended_Value(" GreEn  ")     = green
```

'Extended_Width

Usage:

X'Extended_Width

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameter Descriptions:

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Examples:

Suppose the following types were declared:

```
type X is (red, green, blue, purple);
type Z is (X1, X12, X123, X1234);
```

Then the following would be true:

```
X'Extended_Width     = 6  -- "purple"
Z'Extended_Width     = 5  -- "X1234"
```

3.8.2.2.3. Floating Point Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)                    = " 5.0000E+00"
X'Extended_Image(5.0,1)                  = "5.0000E+00"
X'Extended_Image(-5.0,1)                 = "-5.0000E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Item	A parameter of the predefined type string: it is passed to the function. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Digits

Usage:

X'Extended_Digits(Base)

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter Descriptions:

Base	The base that the subtype is defined in. If no base is specified, the default (10) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Digits     = 5
```

3.8.2.2.4. Fixed Point Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)                  = " 5.00E+00"
X'Extended_Image(5.0,1)                = "5.00E+00"
X'Extended_Image(-5.0,1)               = "-5.00E+00"
X'Extended_Image(5.0,2,0)              = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)            = " 5.0"
X'Extended_Image(5.0,2,0,0,2)          = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)     = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)     = "2#1.1#E+02"
```

'Extended_Value

Usage:

X'Extended_Value(Image)

Returns the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeros, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Image	Parameter of the predefined type string. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")        = 5.0
X'Extended_Value("0.5E1")      = 5.0
X'Extended_Value("2#1.01#E2")  = 5.0
```

'Extended_Fore

Usage:

X'Extended_Fore(Base,Based)

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameter Descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Fore                 = 3  -- "-10"
X'Extended_Fore(2)              = 6  -- " 10001"
```

'Extended_Aft

Usage:

X'Extended_Aft(Base,Based)

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter Descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Aft      = 1  -- "1" from 0.1
X'Extended_Aft(2)   = 4  -- "0001" from 2#0.0001#
```

3.8.3. Package System. The current specification of package System is provided below.

```
with Unchecked_Conversion;

package System is

   --======================
   -- CUSTOMIZABLE VALUES
   --======= ==============

   type Name    is (TeleGen2);

   System_Name  : constant name := TeleGen2;

   Memory_Size  : constant := (2 ** 31) -1; --Available memory, in storage units
   Tick         : constant := 1.0 / 100.0;  --Basic clock rate, in seconds



   --=========================================================
   -- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
   --=========================================================

   -- See Table 8-2 for the values for attributes of
     -- types Float and Long_Float

   Storage_Unit : constant := 8;
   Min_Int      : constant := -(2 ** 31);
   Max_Int      : constant := (2 ** 31) - 1;
   Max_Digits   : constant := 15:
   Max_Mantissa : constant := 31;
   Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);


   subtype Priority is Integer Range 0 .. 63;



   --=========================
   -- ADDRESS TYPE SUPPORT
   --=========================

   type Memory is private;
   type Address is access Memory;
     --
     -- Ensures compatibility between addresses and access types.
     -- Also provides implicit NULL initial value.


   Null_Address: constant Address := null;
```

```
   --
   -- Initial value for any Address object


   type Address_Value is range -(2**31)..(2**31)-1;
   --
   -- A numeric representation of logical addresses for use in address clauses


   Hex_80000000 : constant Address_Value := - 16#80000000#;
   Hex_90000000 : constant Address_Value := - 16#70000000#;
   Hex_A0000000 : constant Address_Value := - 16#60000000#;
   Hex_B0000000 : constant Address_Value := - 16#50000000#;
   Hex_C0000000 : constant Address_Value := - 16#40000000#;
   Hex_D0000000 : constant Address_Value := - 16#30000000#;
   Hex_E0000000 : constant Address_Value := - 16#20000000#;
   Hex_F0000000 : constant Address_Value := - 16#10000000#;
   --
   -- Define numeric offsets to aid in Address calculations
   -- Example:
   --   for Hardware use at Location (Hex_F0000000 + 16#2345678#);


   function Location is new Unchecked_Conversion (Address_Value, Address);
   --
   -- May be used in address clauses:
   --
   --   Object: Some_Type:
   --   for Object use at Location (16#4000#);


   function Label (Name: String) return Address;
   pragma Interface (META, Label);
   --
   -- The LABEL meta-function allows a link name to be specified as address
   -- for an imported object in an address clause:
   --
   --  Object: Some_Type;
   --  for Object use at Label("OBJECT$$LINK_NAME");
   --
   -- System.Label returns Null_Address for non-literal parameters.




   --============================
   -- ERROR REPORTING SUPPORT
   --============================
```

```
procedure Report_Error;
pragma Interface (Assembly, Report_Error):
pragma Interface_Information (Report_Error. "REPORT_ERROR");


--
-- Report_Error can only be called in an exception handler and provides
--  an exception traceback like tracebacks provided for unhandled
--  exceptions
--

--==============
-- CALL SUPPORT
--==============

 type Subprogram_Value IS
 record
   Proc_addr    : Address;
   Parent_frame : Address;
 end record;

 --
 -- Value returned by the implementation-defined 'Subprogram_Value
 -- attribute. The attribute is not defined for subprograms with
 -- parameters, or functions.


private
  type Memory is
  record
   null;
  end record:

end System;
```

3.8.3.1. System.Label The System.Label meta-function is provided to allow users to address objects by a linker-recognized label name. This function takes a single string literal as a parameter and returns a value of System.Address. The function simply returns the run-time address of the appropriate resolved link name, the primary purpose being to address objects created and referenced from other languages.

- When used in an address clause, System.Label indicates that the Ada object or subprogram is to be referenced by a label name. The actual object must be created in some other unit (normally by another language), and this capability simply allows the user to import that object and reference it in Ada.

- When used in an expression, System.Label provides the link time address of any name; a name that might be for an object, a subprogram, etc.

3.8.3.2. System.Report_Error Report_Error can only be called from within an exception handler. This routine displays the normal exception traceback information to standard output. It is essentially the same traceback that could be obtained if the exception were unhandled and propagated out of the program, but the user may want to handle the exception and still display this information. The user may also want to use this capability in a user handler at the end of a task (since those exceptions will not be propagated to the main program). Note that the user can also get this capability for all tasks using the -X binder switch.

For details on the output, see the programming guide chapter in this volume, section "Exception Handling."

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	199 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	199 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except	100 * 'A' & '3' & 99 * 'A'

Name and Meaning	Value
for a character near the middle.	
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	100 * 'A' & '4' & 99 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	197 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	195 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 100 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 99 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	180 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	TELEGEN2
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	1000
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	63
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHAR*`/%
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NONAME/DIRECTORY

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 195 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 193 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 198 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	TELEGEN2
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2147483647
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	TELEGEN2
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. -- 63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is

expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 9C)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

References and page numbers in this appendix are consistent with compiler documentation and not with this report.

3.2. Ada Source Code Preparation

The source for an Ada compilation consists of one or more Ada compilation units (as defined in the LRM, Section 10.1) contained in a single VAX/VMS text file. This text file may be prepared by using any of the standard VAX/VMS text editors (e.g., EDT, EVE). The source file may have any legal VAX/VMS file name. The default file type for Ada source files is ".ADA".

3.3. Running the Compiler

After access to TeleGen2 has been established (Section 1.6) and a library has been created (Section 2.4), the Ada compiler can be used. The syntax for invoking the compiler is:

$ TSADA/E68/ADA{<qualifier>} <file_spec>{,<file_spec>}

where:

<qualifier> is one of the qualifiers available for the compiler.

<file_spec> is one in a possible series of file specifications, separated by commas, indicating the unit(s) to be compiled. If /INPUT_LIST is used, <file_spec> is interpreted as a file containing a list of files to be compiled. The default source file type is ".ADA", and the default list file type is ".LIS." A file name may be qualified with a location in standard VAX/VMS format. A source or input list file may reside on any directory in the system.

The default qualifier settings are designed to allow for the simplest and most convenient use of the compiler. For most applications, no additional qualifier setting need be specified. However, optional qualifiers are provided to:

- Specify maximum errors/warnings before the compiler aborts (/ABORT_COUNT).
- Create an executable program (/BIND).
- Specify the number of source lines listed around each syntactic or semantic error (/CONTEXT).
- Specify the target CPU type (/CPU_NAME).
- Enable the generation of symbolic information for the Source Level Debugger (/DEBUG).
- Select the type of support for floating point operations (/FP_INLINE).
- Compile a list of source files in one compiler invocation (/INPUT_LIST).
- Specify a library other than the default LIBLST.ALB (/LIBFILE and /TEMPLIB).
- Control the generation of source and assembly code listing files (/LIST and /MACHINE_CODE).
- Request compilation progress messages (/MONITOR).
- Restrict compilation to syntactic and semantic analysis (/NOOBJECT).
- Optimize the generated code (/OPTIMIZE).
- Place execution profile code in the generated object code (/PROFILE).
- Delete unneeded intermediate code files after compilation (/SQUEEZE).

- Causes unhandled exceptions in tasks to be reported in the same manner as those that occur in the main program (/SHOW_TASK_EXCEPTION).
- Sets the default amount of stack to allocate from the Ada heap for each task (/TASK_STACK_SIZE).
- Specifies the amount of additional stack space to allocate (in addition to /TASK_STACK_SIZE) for each task (/STACK_GUARD_SIZE).
- Suppress selected run-time checks, source line references, and subprogram name information in generated object code (/SUPPRESS).
- Update the sublibrary after each unit when compiling multiple units (/UPDATE).

These command qualifiers and their default values are further described in detail in Section 3.4.

3.3.1. Multi-Unit Compilation. The compiler may be invoked for multiple sources by specifying each source file on the command line or by using the /INPUT_LIST qualifier. The effect is the same in either case. Files are specified on the command line as follows:

```
$ TSADA/E68/ADA CALC_ARITH,CALC_MEM,[CIOSRC]CALC_IO
```

Functionally, each file listed is treated as a separate compilation. However, the overall compilation rate will be higher than the corresponding separate compilations, as the compiler is only initialized once, and the Ada library is only opened once.

When compiling a list of source files, compilation errors will have two effects. First, the working sublibrary will only be updated for units that compile successfully, and are not in a source file containing a unit that failed. Second, after an error, subsequent units in the source file will be compiled for syntactic and semantic errors only (similar in action to the /NOOBJECT qualifier). Normal compilation will resume with the next source file. The Ada library is updated after each source is compiled. This feature may be disabled by specifying the /NOUPDATE qualifier. In this case, the library is only updated if all sources have compiled successfully.

3.3.2. Specifying a Library. As described in the previous chapter on the Library Manager Tools, the compiler must use a library. As a default, the compiler will use the library file named LIBLST, with a file type of .ALB, in the current working directory. Alternatively, the library may be specified by using either the /LIBFILE or the /TEMPLIB qualifiers described in Section 2.3.2, or by using the the LIBLST logical name as described in Section 2.3.2.3.

The compiler will place the compilation unit information in the working sublibrary of the specified library. If the compiler cannot find the specified library or any of its sublibraries, an error message is issued.

3.3.3. Compiling the Main Program Unit. The main program to be executed must be a parameterless procedure or a parameterless function that returns a scalar value. When the main program is a function, the treatment of the returned value is determined by the installer (see Chapter 7). The compilation unit to be executed as a main program is bound during the compilation process using the /BIND qualifier or after compilation using the TSADA/E68/BIND command (see Section 3.6).

3.3.4. Temporary File Names. Two temporary files are generated by the compiler and deleted when the compilation is complete. These files are created in the same directory as the user's working sublibrary and are named:

Table 3-1. Compiler Command Qualifiers.

Qualifier Name	Action	Default
/ABORT_COUNT=<value>	Specify maximum errors/warnings.	999
/[NO]BIND [=<main_unit>]	/BIND runs the Binder on unit being compiled or on unit specified.	/NOBIND
/CONTEXT =<value>	Request <value> context lines around each error in error listing.	1
/CPU_NAME=[MC68000 | MC68008 | MC68010 | MC68012 | MC68020]	Specify the target CPU type.	MC68000
/[NO]DEBUG	Compile for debugging.	/NODEBUG
/[NO]FP_INLINE	/FP_INLINE enables inline floating point instructions for the MC68020. /NOFP_INLINE enables CGS calls.	/FP_INLINE (when /CPU_NAME= MC68020)
/INPUT_LIST	Input file contains names of files to be compiled, not Ada source.	File contains Ada source.
/LIBFILE=<file_spec>	Specify name of library file.	LIBLST.ALB
/[NO]LIST [=<file_spec>]	/LIST creates a listing file whose default name is <source_file_name>.LIS, or <file_spec>.LIS, if specified.	/NOLIST
/[NO]MACHINE_CODE [=<file_spec>]	Requests a macro assembly listing, which is sent to <comp_unit>.SRC or to <file_spec>, if specified.	/NOMACHINE_CODE
/[NO]MONITOR	/MONITOR requests progress messages.	/NOMONITOR
/[NO]OBJECT	/NOOBJECT restricts compilation to syntactic and semantic analysis.	/OBJECT
/[NO]OPTIMIZE [=(<option>{,<option>})] [<qualifier>]	/OPTIMIZE causes Optimizer to be run on unit(s) being compiled.	/NOOPTIMIZE
/[NO]PROFILE	/PROFILE causes execution profile code to be output in generated object code.	/NOPROFILE
/[NO]SQUEEZE	/SQUEEZE deletes unneeded intermediate unit information after compilation.	/SQUEEZE (/NOSQUEEZE if /DEBUG or /NOOBJECT)
/[NO]SUPPRESS (=<option>{,<option>})	/SUPPRESS suppresses selected run-time checks, names, and/or source line references in generated object code.	/NOSUPPRESS
/TEMPLIB =(<sublib>{,<sublib>})	Specify a temporary library containing listed sublibraries.	None.
/[NO]UPDATE	Update the working sublibrary after each successful compilation.	/UPDATE

In addition to the names of the source files, the input list may contain blank lines and comments in Ada syntax, i.e., all text on a line including and following the comment marker "--" will be ignored. Thus, an equivalent of the example above is a file with the contents:

3.6. Creating Linkable Objects

The object code files generated by the compiler are TeleSoft-defined Object Form files stored in the Ada library. These files must be bound to create a linkable object. The Binder program generates the code needed to elaborate the components in a consistent order.

Before the program can be bound and linked:

- All of the required compilation units must have been successfully compiled in the correct order and must be present in the library.
- The compilation unit that is to serve as the main program must be a parameterless procedure or a parameterless function that returns a scalar value.
- The main program unit *must be located in the working sublibrary.* If this is not the case, reorder the sublibraries in the library file or use TSADA/E68/MOVE or TSADA/E68/COPY to move/copy the unit to the working sublibrary.

If these conditions are met, the user can proceed to bind and link the program.

The general form of the Binder command is:

$ TSADA/E68/BIND{<qualifier>} <main_unit_name>

where:

<qualifier> is one of the qualifiers provided for this command as described below.

<main_unit_name> is the Ada name of the subprogram that is the main program unit (not the name of the source file).

Default settings for the qualifier values were chosen for the simplest and most convenient use of the Binder. For most applications, no additional qualifiers are required. Optional qualifiers are provided to:

- Specify the target CPU type (/CPU_NAME).
- Select the type of support for floating point operations (/FP_INLINE).
- Specify a library other than the default LIBLST.ALB (/LIBFILE and /TEMPLIB).
- Request binding progress messages (/MONITOR).
- Bind the program for use with the Profiler (/PROFILE).
- Set the depth of the run-time exception traceback report (/TRACEBACK).

The Binder can be invoked from standard VAX/VMS command files. On systems with moderate to heavy user loading, it is recommended that binds be performed from command files submitted as batch jobs. See Section 6.1.1 for a discussion of the use of batch queues.

When the Binder is executed, the following steps are performed. First, the library is scanned to find the compilation unit corresponding to <main_unit_name>. This compilation unit is checked to see that it meets the conditions required for a main program; if it does not, an error message is generated and the binding process terminates. Next, the library is scanned to find all members of the main program's extended family, i.e., all units required to execute the main program. As each unit is found, its name and location in the library is noted. If during this process any required units are missing or obsolete, an error message will be generated and the binding process will terminate. If all required units are found and are current, the elaboration code is generated and stored in the Ada library. The program is then ready for linking with the TeleSoft Linker.

If the code to be bound contains one or more units that were compiled with the /PROFILE qualifier, then the /PROFILE qualifier must be supplied to the Binder. See Chapter 12 for a description of the Profiler.

3.6.1. Binder Command Qualifiers. The Binder command qualifiers conform to standard VAX/VMS format and usage. The following sections describe the syntax and semantics of these qualifiers. A summary of these qualifiers is contained in Table 3-6.

Table 3-6. Binder Command Qualifiers.

Qualifier Name	Action	Default
/CPU_NAME=[MC68000 | MC68008 | MC68010 | MC68012 | MC68020]	Specify the target CPU type.	MC68000
/[NO]FP_INLINE	/FP_INLINE enables inline floating point instructions for the MC68020. /NOFP_INLINE enables CGS calls.	/FP_INLINE(when /CPU_NAME=MC68020)
/LIBFILE =<file_spec>	Specify name of library file.	LIBLST.ALB
/[NO]MONITOR	/MONITOR requests progress messages.	/NOMONITOR
/[NO]PROFILE	/PROFILE binds for Profiler use.	/NOPROFILE
/[NO]SHOW_TASK_EXCEPTION	Allows unhandled exceptions within tasks to be reported.	/NOSHOW_TASK_EXCEPTION
/STACK_GUARD_SIZE =<bytes>	Specifies additional task stack space.	256 bytes
/TASK_STACK_SIZE =<bytes>	Specifies the default task stack size.	4096 bytes
/TEMPLIB =(<sublib>{,<sublib>})	Temporary list of sublibraries.	None.
/TRACEBACK =<#_levels>	Set the depth of exception traceback report.	15 levels.

3.6.1.1. Specifying the Target Machine: /CPU_NAME. This qualifier specifies the target CPU in the MC680X0 family and allows exploitation of some of the instruction set additions that have occurred in recent additions to the family. The format of this qualifier is:

/CPU_NAME[=MC68000 | MC68008 | MC68010 | MC68012 | MC68020]

The default is MC68000.

The /CPU_NAME value **must** correspond to the standard libraries and CGS environment modules used, as summarized below:

4.2.1.7. Elimination of Unused Subprograms. If a complete load module is being created, the Linker automatically eliminates Ada subprograms that are not used in the call graph of the main program being linked.

4.2.2. Using the Ada Linker

4.2.2.1. Linker Command Syntax. The VMS command line for the Ada Linker is:

$ TSADA/E68/LINK{<qualifier>} [<compilation_unit_name>]

where:

<qualifier> is none or more of the command line qualifiers listed in Table 4-1.

<compilation_unit_name> is an optional command line parameter indicating the name of the Ada compilation unit to be linked as a main program. The compilation unit must have been bound as a main program prior to linking. If the name of the unit is not provided on the command line, the unit is specified using the INPUT option in an options file.

Linker directives are communicated to the Linker as qualifiers on the VMS command line or as options entered via an options file or SYS$INPUT. Command line qualifiers are useful for controlling options that a user is likely to change often. The default qualifier settings are designed to allow for the simplest and most convenient use of the Linker.

Command line qualifiers and parameters enable the user to:

- Specify the name and format of the linked output file (/LOAD_MODULE. /OBJECT_FORM, /EXECUTE_FORM, /SRECORDS, /OASYS, /IEEE).
- Control the generation and format of listing map files produced by the Linker (/MAP, /IMAGE, /LOCALS, /EXCLUDED, /WIDTH, and /LINES_PER_PAGE).
- Specify an options file (/OPTIONS).
- Specify the starting memory location for the linked output (/BASE).
- Specify the library file containing the components to be linked (/LIBFILE or /TEMPLIB).
- Control the output of debug symbol information for debugging (/DEBUG).
- Monitor the linking process (/MONITOR).

More complicated Linker options, such as the specification of memory locations for specific portions of the code or data for a program, are input via options in a Linker **options file.** Linker options may be used to:

- Specify the compilation units to be used as input to the Linker, the library search paths, and the usage of the input files (INPUT).
- Specify the name and format of the linked output file (OUTPUT).
- Control the generation and format of listing map files produced by the Linker (MAP).
- Specify the location of named memory regions and reserved memory regions in physical memory (REGION).
- Specify the location of control sections in physical memory (LOCATE).
- Define symbol values (DEFINE).
- Specify the target machine on which the output is to be executed (TARGET).

4.2.2.2. Obtaining Lists of Units to be Linked. A list of all Ada compilation units that will be included in a complete link may be obtained by using the library listing command:

```
$ TSADA/SHOW/ELABORATION/NAME=<main_unit>
```

where:

<main_unit> is the name of the main program unit.

Library listing commands are fully described in Section 2.5. The list of included units is also output in the link map.

Table 4-1. Linker Command Line Qualifiers.

Qualifier	Action	Default
/BASE =<address>	Specify start location.	0
/[NO]DEBUG	Output debug information.	/NODEBUG
/EXECUTE_FORM	Produce EF load module format.	/EXECUTE_FORM
/IEEE	Produce IEEE load module format.	Execute Form.
/LIBFILE =<file_spec>	Specify name of library file.	LIBLST.ALB
/LOAD_MODULE [=<file_spec>]	Specify load module output.	/LOAD_MODULE
/[NO]MAP [=<file_spec>] [/[NO]EXCLUDED] [/[NO]IMAGE] [/LINES_PER_PAGE =<value>] (>10) [/[NO]LOCALS] [/WIDTH=<132 | 80>]	Control output of a link map.	/NOMAP /NOEXCLUDED /NOIMAGE 50 /NOLOCALS 132
/[NO]MONITOR	Display Linker progress messages.	/NOMONITOR
/OBJECT_FORM [=<library_ component_name>]	Produce linked OF module output.	None.
/[NO]OPTIONS [=<file_spec>]	Designate options file.	/NOOPTIONS
/SRECORDS	Produce S-Records load module format.	Execute Form.
/TEMPLIB =(<sublib>{,<sublib>})	Temporary list of sublibraries.	None.

Table 4-2. Linker Options and their Qualifiers.

DEFINE /<symbol_name>=<value> [/ADDRESS]	-- Specify link-time values for symbols.
EXIT	-- Terminate options list.
INPUT [/MAIN | */SPEC* | /BODY | /OFM] [/EXPORT_DEFINITIONS] [/PHANTOM | /WORKING_SUBLIB] [/NOSEARCH] <library_component_name>	-- Identify object modules to be linked and specify the search path.
LOCATE [/CONTROL_SECTION=CODE | DATA | CONSTANT] [/COMPONENT_NAME=<library_component_name> [*/SPEC* | /BODY | /OFM]] [/AT=<address>] [/IN=<region_name>] [/AFTER=<control_section_name> | <library_component_name>] [/ALIGNMENT=<value>]	-- Specify addresses for control sections.
MAP [/[*NO*]IMAGE] [/[*NO*]LOCALS] [/[*NO*]EXCLUDED] [/WIDTH=<*132* | 80>] [/LINES_PER_PAGE=<value>] (*50*) [<file_spec>]	-- Control link map generation.
OUTPUT [*/COMPLETE* | /INCOMPLETE] [*/LOAD_MODULE*]=<file_spec>] [/OBJECT_FORM[=<library_component_name>]]	-- Specify complete or incomplete output and its format.
QUIT	-- Abandon link operation.
REGION /LOW_BOUND=<address> /HIGH_BOUND=<address> [/UNUSED] [<region_name>]	-- Define and name memory regions.
TARGET <*MC68000* | MC68010 | MC68020>	-- Specify target processor.

4.2.2.5.3. Specifying the Input: INPUT. This option specifies the name of the Ada library component to be linked, its usage, and its search path. Multiple INPUT options may be used.

By default, the object associated with the specified name is included in the linked output. If the link is incomplete, the symbols defined in the OF modules will not be exported as global definitions. The option line qualifier /EXPORT_DEFINITIONS may be used to override this default. In complete linkage, all references are to be resolved, so the Linker does not export global definitions and ignores this qualifier. The option line qualifiers /MAIN and /PHANTOM may be used to specify special usage for the input OF modules.

The use of an INPUT option does not prevent an *unreferenced* Ada unit from being excluded from the module because of unused subprogram elimination. If an unreferenced unit